Chapter 4.
Classes ATTRIB_HH Aa thru Dz

Topic: Ignore

The class interface is a set of C++ classes, including their public and protected data and methods (member functions), that an application can use directly to interact with ACIS. Developers may also derive their own classes from these classes to add application–specific functionality and data.

This chapter describes the classes for the Healing Component. It contains an alphabetical list of reference templates that describe each class. Refer to the *3D ACIS Online Help User's Guide* for a description of the fields in the reference template.

ATTRIB_HH

Class: Healing, SAT Save and Restore

Purpose: Defines the owning organization for other HEAL attribute classes.

Derivation: ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: "attrib_HH"

Filename: heal/healhusk/attrib/gssl.hxx

Description: ATTRIB_HH is the organizational class from which the other HEAL attribute classes are derived. Its sole purpose is to identify its children classes as belonging to HEAL, and so adds no new data or methods to those of the ACIS base attribute class, ATTRIB.

Limitations: None

References: None

Data:

None

Constructor:

```
public: ATTRIB_HH::ATTRIB_HH (
    ENTITY*                 // owning entity
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_HH::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH::~ATTRIB_HH ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_HH::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int ATTRIB_HH::identity (
    int                     // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_LEVEL.

```
public: virtual logical
    ATTRIB_HH::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: void ATTRIB_HH::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: virtual const char*
    ATTRIB_HH::type_name () const;
```

Returns the string “attrib_HH”.

Related Fncs:

is_ATTRIB_HH

ATTRIB_HH_AGGR

Class: Healing, SAT Save and Restore

Purpose: Base HEAL aggregate attribute class.

Derivation: ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_body_attribute”

Filename: heal/healhusk/attrib/at_aggr.hxx

Description: ATTRIB_HH_AGGR is the base aggregate attribute class from which other HEAL aggregate attribute classes are derived. Aggregate attributes are attached to the body being healed to store information about each phase or subphase of the healing process. Aggregate attributes also manage the individual attributes attached to entities of the body during healing.

Limitations: None

References: None

Data:

```
protected enum MODULE_HEAL_STATUS m_module_state;
```

Healing status.

Constructor:

```
public: ATTRIB_HH_AGGR::ATTRIB_HH_AGGR (
    BODY* owner             // owning body to heal
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_HH_AGGR::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual
    ATTRIB_HH_AGGR::~ATTRIB_HH_AGGR ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR(...) then later x–>lose.)

Methods:

```
public: virtual void ATTRIB_HH_AGGR::analyze ();
```

This is a virtual function that must be implemented by every class derived from this class. Analyzes the body and compute the best options and tolerances for a particular healing phase/subphase.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR::attach_attrib (
    ENTITY*                 // entity to attach to
    );
```

This is a virtual function that must be implemented by every class derived from this class. Attaches an individual entity–level attribute to the given entity. This method chains individual attributes.

```
public: virtual void ATTRIB_HH_AGGR::calculate ();
```

This is a virtual function that must be implemented by every class derived from this class. Performs the calculations for a particular healing phase or subphase and stores all the recommended changes in individual attributes.

```
public: virtual void ATTRIB_HH_AGGR::cleanup ();
```

This is a virtual function that must be implemented by every class derived from this class. Removes all individual attributes.

```
public: virtual void ATTRIB_HH_AGGR::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual void ATTRIB_HH_AGGR::detach_attrib (
    ENTITY*                 // owning entity
    );
```

This is a virtual function that must be implemented by every class derived from this class. Removes the attribute for the entity.

```
public: virtual void ATTRIB_HH_AGGR::entity_list (
    ENTITY_LIST&            // list of entities
    ) const;
```

This is a virtual function that must be implemented by every class derived from this class. Gets all the entities chained to the body. These are entities to which individual attributes are attached.

```
public: virtual void ATTRIB_HH_AGGR::fix ();
```

This is a virtual function that must be implemented by every class derived from this class. Applies (fixes) all the changes that are stored in individual attributes for a particular healing phase/subphase to the body.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR::get_attrib (
    ENTITY*                 // owning entity
    ) const;
```

This is a virtual function that must be implemented by every class derived from this class. Gets the attribute for the corresponding entity.

```
public: virtual int ATTRIB_HH_AGGR::identity (
    int                     // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_LEVEL.

```
public: virtual logical
    ATTRIB_HH_AGGR::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: enum MODULE_HEAL_STATUS
ATTRIB_HH_AGGR::module_state ();
```

Gets the current status of healing.

```
public: virtual logical
    ATTRIB_HH_AGGR::pattern_compatible () const;
```

Returns TRUE if this is pattern compatible.

```
public: virtual void ATTRIB_HH_AGGR::print (
    FILE*                   // file to print to
    );
```

This is a virtual function that must be implemented by every class derived from this class. Prints the results of the operations in a particular healing phase/subphase.

```
public: virtual void ATTRIB_HH_AGGR::print_analyze (
    FILE*                   // file to print to
    );
```

This is a virtual function that must be implemented by every class derived from this class. Prints the results of the analyze stage of a particular healing phase/subphase.

```
public: virtual void
    ATTRIB_HH_AGGR::print_calculate (
    FILE*                   // file to print to
    );
```

This is a virtual function that must be implemented by every class derived from this class. Prints the results of the calculate stage of a particular healing phase/subphase.

```
public: virtual void
    ATTRIB_HH_AGGR::print_check (
    FILE*                   // file to print to
    );
```

This is a virtual function that must be implemented by every class derived from this class. Prints the results of the check stage of a particular healing phase/subphase.

```
public: virtual void ATTRIB_HH_AGGR::print_fix (
    FILE*                   // file to print to
    );
```

This is a virtual function that must be implemented by every class derived from this class. Prints the results of the fix stage of a particular healing phase.

```
public: void ATTRIB_HH_AGGR::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: virtual void
    ATTRIB_HH_AGGR::set_heal_status (
    enum MODULE_HEAL_STATUS state   // status
    );
```

Sets the current status of healing.

```
public: virtual const char*
    ATTRIB_HH_AGGR::type_name () const;
```

Returns the string “aggregate_body_attribute”.

Related Fncs:

find_aggr_attrib, is_ATTRIB_HH_AGGR

ATTRIB_HH_AGGR_ANALYTIC

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the analytic solver subphase of geometry building.

Derivation: ATTRIB_HH_AGGR_ANALYTIC : ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_analytic_solver_attribute”

Filename: heal/healhusk/attrib/hanalsol.hxx

Description: The ATTRIB_HH_AGGR_ANALYTIC class is attached to the body to be healed. Is is used by the analytic solver subphase of the geometry building healing phase. The analytic solver subphase attempts to heal all edges and vertices shared by analytic surfaces.

Limitations: None

References: None

Data:

```
protected bhl_analytic_solver_results results;
```
Structure containing the results of the analytic solver subphase.

```
protected bhl_geombld_options geom_bld_opt;
```
Options for the analytic solver.

```
protected HH_Tangent_Analytic_Snapper snapper;
```
Graph based tool for the analytic solver.

```
protected HH_Unstable_Vertex_Solver uvsolver;
```
Unstable vertex solver.

```
protected Unstable_Vertex_Solver
    unstable_vertex_solver;
```
Unstable vertex solver used by the analytic solver.

```
protected logical save_sw;
```
For future use.

```
public HH_SurfSnap_Node_Type HH_SurfSnap_Type;
```
Identifies node of type HH_SurfSnap.

```
public HH_SurfSnap_Node_Type HH_Unstable_Snap_Type;
```
Identifies node of type HH_Unstable_SurfSnap.

```
public logical bhl_do_uv_tan_graph;
```
Option for setting uv_tan_grph solver.

```
public double bhl_snap_doub_tol;
```
The maximum tolerance for scaling beyond which no analytic face is scaled.

```
public double bhl_snap_pos_tol;
```
The maximum tolerance for translation beyond which no analytic face is translated.

```
public double bhlabs;
```
Minimum translation deviation for the analytic solver.

```
public double bhlmch;
```
Minimum scale deviation for the analytic solver.

```
public double bhlnor;
```
Minimum rotation deviation for the analytic solver.

```
public hh_anal_solv_options anal_options;
```
A Structure which contains options and tolerances for their analytic solver module.

```
public int bhl_anal_sol_coin_resolved;
```
Number of coincident faces resolved.

```
public int bhl_anal_sol_coin_unresolved;
```
Number of coincident faces unresolved.

```
public int bhl_anal_sol_degree;
```
The degree of the analytic tangency graph.

```
public int bhl_anal_sol_tang_resolved;
```
Number of analytic tangencies resolved.

```
public int bhl_anal_sol_tang_unresolved;
```
Number of analytic tangencies un–resolved.

```
public int bhl_anal_sol_unst_vert;
```
Number of unstable vertices fixed.

```
public int bhl_anal_solver_stage;
```
A flag to denote the current stage of the analytic solver.

```
public logical bhl_snap_main;
```
Option to denote if the main part of the analytic solver is on/off.

```
public logical bhl_snap_check;
```
Option to denote if the simple snap correction at the end of the analytic solver is on or off.

```
public logical bhl_unstable_vert;
```
Option to denote if the unstable vertex part of the analytic solver is on or off.

```
public int m_analytic_geom_worsened;
```
Option to denote if the geometry has worsened.

Constructor:

```
public: ATTRIB_HH_AGGR_ANALYTIC::
    ATTRIB_HH_AGGR_ANALYTIC (
    BODY* b                  // owning body
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_ANALYTIC(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_ANALYTIC::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_ANALYTIC::
    ~ATTRIB_HH_AGGR_ANALYTIC ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_ANALYTIC(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_ANALYTIC::analyze ();
```

Performs the analyze stage of the analytic solver subphase of geometry building. Analyzes the body and compute the best options and tolerances for the analytic solver subphase.

```
public: virtual void
    ATTRIB_HH_AGGR_ANALYTIC::calculate ();
```

Performs the calculate stage of the analytic solver subphase of geometry building and stores all the recommended changes in individual attributes. This method finds how each face should be snapped so as to make all the tangencies valid, and stores the results in the attributes.

```
public: const bhl_analytic_solver_results&
    ATTRIB_HH_AGGR_ANALYTIC::calculation_results ()
    const;
```

Returns the results of the analytic solver subphase.

```
public: bhl_analytic_solver_results*
ATTRIB_HH_AGGR_ANALYTIC::
calculation_results_for_change ();
```

Fills in the calculate results structure.

```
protected: virtual void
    ATTRIB_HH_AGGR_ANALYTIC::cleanup ();
```

Removes all individual entity–level attributes from the entities of the owning body.

```
protected: void
    ATTRIB_HH_AGGR_ANALYTIC::coincident_solver ();
```

Executes the coincident solver, which snaps “almost coincident” faces.

```
public: virtual void
    ATTRIB_HH_AGGR_ANALYTIC::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: logical
    ATTRIB_HH_AGGR_ANALYTIC::do_analytic () const;
```

Returns FALSE if the analytic solver subphase has been turned off, otherwise, returns TRUE.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::fill_results (
    bhl_analytic_solver_results&,    // results
    bhl_geombld_options&             // options
    );
```

Gathers various healing results from globals and options into the results structure.

```
public: bhl_geombld_options*
    ATTRIB_HH_AGGR_ANALYTIC::
    geombld_results_for_change ();
```

Returns analytic solver results.

```
public: double
    ATTRIB_HH_AGGR_ANALYTIC::get_rot_tol () const;
```

Returns the rotation tolerance value.

```
public: double
    ATTRIB_HH_AGGR_ANALYTIC::get_scale_tol () const;
```

Returns the scale tolerance value.

```
public: virtual int
    ATTRIB_HH_AGGR_ANALYTIC::identity (
    int                       // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_ANALYTIC_LEVEL.

```
public: logical ATTRIB_HH_AGGR_ANALYTIC::
is_analytic_geom_worsened ();;
```

Checks the geometry of the body. Returns TRUE if the geometry has worsened.

```
public: logical ATTRIB_HH_AGGR_ANALYTIC::
    is_analytic_tangency_good ();
```

Checks analytic tangencies of body. Returns TRUE if all analytic tangencies are good.

```
public: virtual logical
    ATTRIB_HH_AGGR_ANALYTIC::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: logical ATTRIB_HH_AGGR_ANALYTIC::
    is_unstable_vertex_good ();
```

Checks unstable vertices of body. Returns TRUE if all unstable vertices are good.

```
public: void
    ATTRIB_HH_AGGR_ANALYTIC::log_analytic_details ();
```

Allows additions to the log list.

```
protected: void ATTRIB_HH_AGGR_ANALYTIC::
    normal_and_scale_solver ();
```

Executes the normal solver and the scale solver.

```
public: virtual logical
    ATTRIB_HH_AGGR_ANALYTIC::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::print (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the analytic solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_ANALYTIC::print_calculate (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the calculate stage of the analytic solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_ANALYTIC::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

if(restore_version_number >= TOL_MODELING_VERSION)

read_logical	This is the save_sw data item.
restore_arcs_nodes	snapper
restore_body	unstable vertex solver
restore_body	new unstable vertex solver

```
public: void
    ATTRIB_HH_AGGR_ANALYTIC::set_do_analytic (
    logical use             // use this subphase flag
    );
```

Sets a flag indicating whether or not the analytic solver subphase needs to be used. If TRUE, the analytic solver subphase is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::set_globals ();
```

Initializes various global variables to zero.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::set_rot_tol (
    double r                // tolerance to use
    );
```

Sets the rotation tolerance to the specified value.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::set_scale_tol (
    double s                // tolerance to use
    );
```

Sets the scale tolerance to the specified value.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::set_tol (
    double t                // tolerance to use
    );
```

Sets the tolerance used by the analytic solver subphase.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::set_tolerances
();
```

Sets the tolerance used by the analytic solver subphase.

```
public: void ATTRIB_HH_AGGR_ANALYTIC::sprint (
    char*                   // character string
    );
```

Prints statistics of the results of the analytic solver subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_ANALYTIC::sprint_calculate (
    char*                   // character string
    );
```

Prints statistics of the results of the calculate stage of the analytic solver subphase to the specified string.

```
public: double ATTRIB_HH_AGGR_ANALYTIC::tol () const;
```

Gets the current value of the analytic solver subphase tolerance.

```
public: virtual const char*
    ATTRIB_HH_AGGR_ANALYTIC::type_name () const;
```

Returns the string “aggregate_analytic_solver_attribute”.

Related Fncs:

find_aggr_analytic, is_ATTRIB_HH_AGGR_ANALYTIC

ATTRIB_HH_AGGR_GEN_SPLINE

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the generic spline solver subphase of geometry building.

Derivation: ATTRIB_HH_AGGR_GEN_SPLINE : ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_advspl_attribute”

Filename: heal/healhusk/attrib/hadvspl.hxx

Description: The ATTRIB_HH_AGGR_GEN_SPLINE class is attached to the body to be healed. Is is used by the generic spline solver subphase of the geometry building healing phase. The generic spline solver attempts to heal generic tangential spline junctions, (e.g., the intersection curve is *not* an isoparametric curve of both splines in the intersection).

Limitations: None

References: None

Data:

```
protected bhl_advanced_spline_solver_results results;
```

Structure containing results of the generic spline solver subphase.

```
protected hh_advspl_options advspl_opt;
```

Structure containing the options used by the generic spline solver subphase. These options can be set and queried using methods of this class.

```
protected logical save_sw;
```
For future use.

Constructor:

```
public:ATTRIB_HH_AGGR_GEN_SPLINE::
    ATTRIB_HH_AGGR_GEN_SPLINE (
    BODY* b                  // owning body to heal
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_GEN_SPLINE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_GEN_SPLINE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_GEN_SPLINE::
    ~ATTRIB_HH_AGGR_GEN_SPLINE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_GEN_SPLINE(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_GEN_SPLINE::analyze ();
```

Performs the analyze stage of the generic spline solver subphase of geometry building. Analyzes the body and compute the best options and tolerances for the generic spline solver subphase.

```
public: virtual void
    ATTRIB_HH_AGGR_GEN_SPLINE::calculate ();
```

Performs the calculate stage of the generic spline solver subphase of geometry building and stores all the recommended changes in individual attributes.

```
public: const bhl_advanced_spline_solver_results
    ATTRIB_HH_AGGR_GEN_SPLINE::calculation_results
    () const;
```

Returns the results of the generic spline solver subphase.

```
public: bhl_advanced_spline_solver_results*
ATTRIB_HH_AGGR_GEN_SPLINE::
calculation_results_for_change ();
```

Fills in the calculate results structure.

```
public: virtual void
    ATTRIB_HH_AGGR_GEN_SPLINE::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: logical ATTRIB_HH_AGGR_GEN_SPLINE::
    do_gen_spline () const;
```

Returns FALSE if the generic spline solver subphase has been turned off, otherwise, returns TRUE.

```
public: virtual int
    ATTRIB_HH_AGGR_GEN_SPLINE::identity (
    int                      // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_GEN_SPLINE_LEVEL .

```
public: virtual logical
    ATTRIB_HH_AGGR_GEN_SPLINE::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical
    ATTRIB_HH_AGGR_GEN_SPLINE::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_HH_AGGR_GEN_SPLINE::print (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the generic spline solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_GEN_SPLINE::print_calculate (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the calculate stage of the generic spline solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_GEN_SPLINE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if(restore_version_number >= TOL_MODELING_VERSION)
    read_logical                This is the save_sw data item.
```

```
public: void
    ATTRIB_HH_AGGR_GEN_SPLINE::set_do_gen_spline (
    logical t               // use this subphase flag
    );
```

Sets a flag indicating whether or not the generic spline solver subphase needs to be used. If TRUE, the generic spline solver subphase is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_GEN_SPLINE::sprint (
    char*                   // character string
    );
```

Prints statistics of the results of the generic spline solver subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_GEN_SPLINE::sprint_calculate (
    char*                   // character string
    );
```

Prints statistics of the results of the calculate stage of the generic spline solver subphase to the specified string.

```
public: virtual const char*
    ATTRIB_HH_AGGR_GEN_SPLINE::type_name () const;
```

Returns the string “aggregate_advspl_attribute”.

Related Fncs:

find_aggr_gen_spline, is_ATTRIB_HH_AGGR_GEN_SPLINE

ATTRIB_HH_AGGR_GEOMBUILD

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the geometry building phase.

Derivation: ATTRIB_HH_AGGR_GEOMBUILD :
ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR :
ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_geombuild_attribute”

Filename: heal/healhusk/attrib/hmaster.hxx

Description: The ATTRIB_HH_AGGR_GEOMBUILD class is attached to the body to be healed. Is is used by the geometry building healing phase.

Limitations: None

References: None

Data:

`protected bhl_anal_geometry_results anal_results;`
Structure containing the results of the geometry building analysis of the input (to be healed) body.

`public HH_Anal_Geombld* geom_analyzer;`
Points to the main routine which does all the analysis of the input BODY. The analysis finds all the bad geometry in the BODY and adds attributes to the bad parts of the model to be used during the repair stage.

```
protected bhl_anal_geometry_results
    output_anal_results;
```
Structure containing the results of the geometry building analysis of the output (healed) body.

`protected bhl_geometry_results results;`
Structure containing the results of the geometry building phase.

`protected hh_geombuild_options geombuild_options;`
Structure containing the options and tolerances used by the geometry building phase.

`protected logical m_do_geombuild_log;`
Option for logging changes to entities in a geombuild module.

`protected logical m_pcurves_already_cleaned;`
Analysis results of the healed body.

```
protected MODULE_HEAL_STATUS
    m_postprocess_heal_state;
```
Flag to denote preprocess stage.

`protected MODULE_HEAL_STATUS m_preprocess_heal_state;`
Flag to denote postprocess stage.

Constructor:

```
public: ATTRIB_HH_AGGR_GEOMBUILD::
    ATTRIB_HH_AGGR_GEOMBUILD (
    BODY* b                  // owning body
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_GEOMBUILD(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_GEOMBUILD::
    ~ATTRIB_HH_AGGR_GEOMBUILD ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_GEOMBUILD(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze ();
```

Performs advanced analysis on the geometry of the body. It checks the body owned by the aggregate attribute and stores results in individual entity–level attributes. This method executes all the various “advanced analysis” methods that are defined in this class for specific entity types (vertices, edges, coedges, loops, faces, shells, lumps, curves, pcurves, and surfaces).

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_coedge ();
```

Performs advanced analysis on the coedges of the body. It checks the coedges of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Does the coedge lie on the corresponding face surface?
– If the coedge contains a pcurve, does the domain of the pcurve correspond with the edge?
– Does the coedge have a partner?
– If the coedge contains a pcurve, is the pcurve within tolerance of the edge?

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_curve ();
```

Performs advanced analysis on the curves of the body. It checks the curves of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Is the curve continuous?
– Is the curve degenerate?
– Is the curve self–intersecting?
– Is the curve periodic?
– Is the curve an approximate curve?

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_edge ();
```

Performs advanced analysis on the edges of the body. It checks the edges of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Check the curve geometry (adv_analyze_curve).
– Determine convexity.
– Check edge length.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_face ();
```

Performs advanced analysis on the faces of the body. It checks the faces of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Check the loops (adv_analyze_loop).
– Check the surface (adv_analyze_surface).
– Check face area.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_loop ();
```

Performs advanced analysis on the loops of the body. It checks the loops of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Is the loop closed?
– Is the loop orientation correct?
– Do the loop's coedges have gaps?
– Does the loop self–intersect?
– Check for correct parameter range of the coedges.
– Check that the coedges lie on the face surface.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_lump ();
```

Performs advanced analysis on the lumps of the body. It checks the lumps of the body owned by the aggregate attribute and stores results in individual entity–level attributes. This checks the shells for closure.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_pcurve ();
```

Performs advanced analysis on the pcurves of the body. It checks the pcurves of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Is the pcurve continuous?
– Is the pcurve degenerate?
– Is the pcurve self–intersecting?
– Does the pcurve lie on the edge?
– Is the pcurve parameter range correct?

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_shell ();
```

Performs advanced analysis on the shells of the body. It checks the shells of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Is the shell closed?
– Check shell orientation.
– Check if shell represents a single volume.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_surface ();
```

Performs advanced analysis on the surfaces of the body. It checks the surfaces of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Is the surface continuous?
– Is the surface degenerate?
– Is the surface self–intersecting?
– Is the surface periodic?

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::adv_analyze_vertex ();
```

Performs advanced analysis on the vertices of the body. It checks the vertices of the body owned by the aggregate attribute and stores results in individual entity–level attributes.

The tests performed include:

– Does the vertex lie on the corresponding edges?
– Do the edges meet at the vertex?
– Does the vertex lie on the corresponding surfaces?

```
public: const bhl_anal_geometry_results
    ATTRIB_HH_AGGR_GEOMBUILD::analysis_results
    () const;
```

Returns the results of geometry building analysis.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::analyze ();
```

Performs the analyze stage of the geometry building phase. Analyzes the body and compute the best options and tolerances for the various subphases of the geometry building phase.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    attach_all_aggr_attribs ();
```

Attaches the aggregate attributes for all the geometry building subphases to the body being healed.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::calculate ();
```

Performs the calculate stage of the geometry building phase and stores all the recommended changes in individual attributes. The new geometry for all the unhealed portions of the body is calculated.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::check (
    bhl_anal_geometry_results*   // struct
        check_results            // containing results
    );
```

Attaches individual geometry building attributes to the body and checks the body for bad geometry.

```
public: logical ATTRIB_HH_AGGR_GEOMBUILD::
    check_discontinuity () const;
```

Gets the option that denotes if checking edge discontinuity is on or off.

```
public: void
    ATTRIB_HH_AGGR_GEOMBUILD::cleanup_pcurves ();
```

Clean up pcurves and make approximate intcurves.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    compute_analytic_rot_tol ();
```

Computes the rotation tolerance value.

```
public: double ATTRIB_HH_AGGR_GEOMBUILD::
    compute_max_spline_tang_tol ();
```

Computes the maximum spline tangent tolerance.

```
public: double ATTRIB_HH_AGGR_GEOMBUILD::
    compute_min_spline_tang_tol ();
```

Computes the minimum spline tangent tolerance.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    compute_spline_tang_tols ();
```

Computes the spline tangent tolerances.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: logical
    ATTRIB_HH_AGGR_GEOMBUILD::do_geombuild () const;
```

Returns FALSE if the geometry building phase has been turned off; otherwise, returns TRUE.

```
public: logical ATTRIB_HH_AGGR_GEOMBUILD::
    do_geombuild_log () const;
```

Toggles module change logging. Returns FALSE if logging has been turned off; otherwise, returns TRUE.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::fix ();
```

Applies (fixes) all the changes that are stored in individual attributes for the geometry building phase to the body. The old geometry is then stored in the attributes.

```
public: const bhl_geometry_results
    ATTRIB_HH_AGGR_GEOMBUILD::geom_results () const;
```

Returns the results of the geometry building phase.

```
public: virtual int
    ATTRIB_HH_AGGR_GEOMBUILD::identity (
    int                       // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_GEOMBUILD_LEVEL.

```
public: void
    ATTRIB_HH_AGGR_GEOMBUILD::initialize_tols (
    );
```

Initializes all the tolerances used in the geometry building phase (including the various geometry building subphases).

```
public: virtual logical
    ATTRIB_HH_AGGR_GEOMBUILD::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: double ATTRIB_HH_AGGR_GEOMBUILD::
    max_spline_tang_tol () const;
```

Returns the maximum spline tangent tolerance.

```
public: double ATTRIB_HH_AGGR_GEOMBUILD::
    min_spline_tang_tol () const;
```

Returns the minimum spline tangent tolerance.

```
public: double
    ATTRIB_HH_AGGR_GEOMBUILD::min_tol () const;
```

Gets the minimum geombuild tolerance.

```
public: const bhl_anal_geometry_results
    ATTRIB_HH_AGGR_GEOMBUILD::
    output_analysis_results () const;
```

Returns the geometry building analysis results for the output body.

```
public: virtual logical
    ATTRIB_HH_AGGR_GEOMBUILD::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: enum MODULE_HEAL_STATUS
    ATTRIB_HH_AGGR_GEOMBUILD::
    postprocess_heal_state ();
```

Gets the value of the flag which denotes the postprocess stage.

```
public: enum MODULE_HEAL_STATUS
    ATTRIB_HH_AGGR_GEOMBUILD::
    preprocess_heal_state ();
```

Gets the value of the flag which denotes the preprocess stage.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::print (
    FILE* fp                // file pointer
    );
```

Prints the geometry building results to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::print_analyze (
    FILE* fp                // file pointer
    );
```

Prints the results of the analyze stage of the geometry building phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::print_calculate (
    FILE* fp                // file to print to
    );
```

Prints the results of the calculate stage of the geometry building phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::print_check (
    FILE* fp                // file to print to
    );
```

Prints the results of the check stage of the geometry building phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::print_fix (
    FILE* fp                // file to print to
    );
```

Prints the results of the fix stage of the geometry building phase to the specified file.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    reset_edges_tangency_details ();
```

Initialize the edge tangency information that specifies whether an edge is a tangent edge or not.

```
public: void
    ATTRIB_HH_AGGR_GEOMBUILD::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    set_check_discontinuity (
    logical val             // on or off
    );
```

Sets the option that denotes if checking edge discontinuity is on or off.

```
public: void
    ATTRIB_HH_AGGR_GEOMBUILD::set_do_geombuild (
    logical use             // use this phase flag
    );
```

Sets a flag indicating whether or not the geometry building phase needs to be used. If TRUE, the geometry building phase is used; otherwise, it is not used.

```
public: void
    ATTRIB_HH_AGGR_GEOMBUILD::set_do_geombuild_log (
    logical                 // toggle logging
    );
```

Used for logging simplification module changes.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    set_max_spline_tang_tol (
    double                  // global tolerance
    );
```

Set the maximum global tolerance value to the input value given.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    set_min_spline_tang_tol (
    double                  // tolerance value
    );
```

Sets the minimum global spline tangent tolerance to the specified value.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::set_min_tol (
    double t                // minimum tolerance
    );
```

Sets the minimum geombuild tolerance.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    set_postprocess_heal_status (
    enum MODULE_HEAL_STATUS state   // status
    );
```

Sets the postprocessor stage flag.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    set_preprocess_heal_status (
    enum MODULE_HEAL_STATUS state   // status
    );
```

Sets the preprocess stage flag.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::set_tang_tol (
    double                  // tangent tolerance
    );
```

Sets the tangent tolerance to the specified value.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::set_tol (
    double t                // tolerance to use
    );
```

Sets the tolerance used by the geometry building phase.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::sprint (
    char*                   // character string
    );
```

Prints the geometry building results to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::sprint_analyze (
    char*                   // character string
    );
```

Prints the results of the analyze stage of the geometry building phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::sprint_calculate (
    char*                    // character string
    );
```

Prints the results of the calculate stage of the geometry building phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::sprint_check (
    char*                    // character string
    );
```

Prints the results of the check stage of the geometry building phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD::sprint_fix (
    char*                    // character string
    );
```

Prints the results of the fix stage of the geometry building phase to the specified string.

```
public: double ATTRIB_HH_AGGR_GEOMBUILD::
    tang_tol () const;
```

Returns the tangent tolerance.

```
public: double
    ATTRIB_HH_AGGR_GEOMBUILD::tol () const;
```

Gets the current value of the geometry building tolerance.

```
public: virtual const char*
    ATTRIB_HH_AGGR_GEOMBUILD::type_name () const;
```

Returns the string “aggregate_geombuild_attribute”.

```
public: void ATTRIB_HH_AGGR_GEOMBUILD::
    update_all_edge_data ();
```

Finds all the edges in the model and computes the min_angle and max_angle data for the edge.

Related Fncs:

find_aggr_geombuild, is_ATTRIB_HH_AGGR_GEOMBUILD

ATTRIB_HH_AGGR_GEOMBUILD_BASE

Class: Healing, SAT Save and Restore

Purpose: Base HEAL aggregate attribute class for the geometry building phase.

Derivation: ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_geombuild_base_attribute”

Filename: heal/healhusk/attrib/aggrgbld.hxx

Description: ATTRIB_HH_AGGR_GEOMBUILD_BASE is the base geometry building aggregate attribute class from which other HEAL aggregate attribute classes used in the geometry building phase are derived. Aggregate attributes are attached to the body being healed to store information about each subphase of the geometry building phase. Aggregate attributes also manage the individual attributes attached to entities of the body during geometry building.

Limitations: None

References: None

Data:

None

Constructor:

```
public: ATTRIB_HH_AGGR_GEOMBUILD_BASE::
    ATTRIB_HH_AGGR_GEOMBUILD_BASE (
    BODY* b                  // owning body to heal
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_GEOMBUILD_BASE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_GEOMBUILD_BASE::
    ~ATTRIB_HH_AGGR_GEOMBUILD_BASE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_GEOMBUILD_BASE(...) then later x–>lose.)

Methods:

```
public: void ATTRIB_HH_AGGR_GEOMBUILD_BASE::
    attach_all_entity_attribs ();
```

Attaches the entity–level attributes for the geometry building phase to all the individual entities of the owning body.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::attach_attrib (
    ENTITY*                 // entity to attach to
    );
```

Attaches an individual entity–level attribute to the given entity. This method chains individual attributes.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::cleanup ();
```

Removes the entity–level attributes from the individual entities of the owning body.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::detach_attrib (
    ENTITY*                 // owning entity
    );
```

Removes the entity–level geometry building attributes for the specified individual entity.

```
public: virtual void ATTRIB_HH_AGGR_GEOMBUILD_BASE::
    detach_redundant_attribs ();
```

Removes unused entity–level attributes from the individual entities of the owning body.

```
public: virtual void
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::entity_list (
    ENTITY_LIST&            // entities in the body
    ) const;
```

Returns the list of entities that have entity–level attributes attached.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::get_attrib (
    ENTITY*                 // entity with attribute
    ) const;
```

Returns the entity–level attribute for the specified entity.

```
public: virtual int
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::identity (
    int                     // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_GEOMBUILD_BASE_LEVEL .

```
public: virtual logical
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::pattern_compatible
() const;
```

Returns TRUE if this is pattern compatible.

```
public: void
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data	This class does not save any data

```
public: virtual const char*
    ATTRIB_HH_AGGR_GEOMBUILD_BASE::
    type_name () const;
```

Returns the string “aggregate_geombuild_base_attribute”.

Related Fncs:

find_aggr_geombuild_base,
is_ATTRIB_HH_AGGR_GEOMBUILD_BASE

ATTRIB_HH_AGGR_ISOSPLINE

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the isospline solver subphase of geometry building.

Derivation: ATTRIB_HH_AGGR_ISOSPLINE :
ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR :
ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_isospline_attribute”

Filename: heal/healhusk/attrib/huvsolv.hxx

Description: The ATTRIB_HH_AGGR_ISOSPLINE class is attached to the body to be healed. Is is used by the isospline solver subphase of the geometry building healing phase. The isospline solver attempts to heal all edges shared by tangential isoparametric surfaces (e.g., the intersection curve is an isoparametric curve of both splines in the intersection).

Limitations: None

References: None

Data:

```
protected bhl_spline_solver_results results;
```
Structure containing results of the isospline solver subphase.

```
protected hh_isospline_options uvspl_opt;
```
Structure containing the options used by the isospline solver subphase. These options can be set and queried using methods of this class.

```
protected logical save_sw;
```
For future use.

Constructor:

```
public: ATTRIB_HH_AGGR_ISOSPLINE::
    ATTRIB_HH_AGGR_ISOSPLINE (
    BODY* b                  // owning body to heal
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_ISOSPLINE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
1public: virtual void
    ATTRIB_HH_AGGR_ISOSPLINE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_ISOSPLINE::
    ~ATTRIB_HH_AGGR_ISOSPLINE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_ISOSPLINE(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_ISOSPLINE::analyze ();
```

Performs the analyze stage of the isospline solver subphase of geometry building. Analyzes the body and compute the best options and tolerances for the isospline solver subphase.

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::analyze_C1 ();
```

Analyzes tangent junction for possible C1 calculation.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    attach_isospline_attribs ();
```

Attaches the isospline attributes for the isospline solver subphase to the body being healed.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    bend_faces_to_iso_vertices ();
```

Used for isospline healing. Bends the entity faces to the isospline vertices.

```
protected: void ATTRIB_HH_AGGR_ISOSPLINE::
    bend_splines_to_strips ();
```

Modifies the associated spline surfaces to match the C1 strip surface.

```
public: virtual void
    ATTRIB_HH_AGGR_ISOSPLINE::calculate ();
```

Performs the calculate stage of the isospline solver subphase of geometry building and stores all the recommended changes in individual attributes.

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::calculate_C1 ();
```

Main function to perform the C1 calculations.

```
public: const bhl_spline_solver_results
    ATTRIB_HH_AGGR_ISOSPLINE::calculation_results
    () const;
```

Returns the results of the isospline solver subphase.

```
public: bhl_spline_solver_results*
    ATTRIB_HH_AGGR_ISOSPLINE::
    calculation_results_for_change ();
```

Fills in the calculate results structure.

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::classify_C1 ();
```

Classifies junctions as possible C1 junctions.

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::compute_C1_ratios ();
```

Computes the required parameters to make a junction C1 continuous.

```
public: virtual void
    ATTRIB_HH_AGGR_ISOSPLINE::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    detach_isospline_attribs ();
```

Removes the isospline attributes from the entity.

```
public: logical
    ATTRIB_HH_AGGR_ISOSPLINE::do_isospline () const;
```

Returns FALSE if the isospline solver subphase has been turned off, otherwise, returns TRUE.

```
public: logical ATTRIB_HH_AGGR_ISOSPLINE::
    do_twist_correction () const;
```

Gets the option to denote if twist correction calculation is on or off.

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::equip_attribs ();
```

Defines the attributes required for the C1 calculations.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    fix_procedural_geometry ();
```

Used for isospline healing. Makes the procedural geometry exact, wherever necessary.

```
protected: logical ATTRIB_HH_AGGR_ISOSPLINE::
    generate_sequence_for_C1 ();
```

Determines the sequence of faces to consider for C1 continuity calculation.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    heal_isospline_edges ();
```

Used for isospline healing. One–by–one, heals all of the isospline edges in the body, using member functions of isospline edge attributes.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    heal_isospline_vertices ();
```

Used for isospline healing. Places all the vertices in the owner body into the best possible location. Bends the faces to the iso vertices (i.e., vertices of the isospline edges).

```
public: virtual int
    ATTRIB_HH_AGGR_ISOSPLINE::identity (
    int                       // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_ISOSPLINE_LEVEL.

```
public: virtual logical
    ATTRIB_HH_AGGR_ISOSPLINE::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
protected: void ATTRIB_HH_AGGR_ISOSPLINE::
    make_boundary_curves_C1 ();
```

Makes the surface boundary curve C1 continuous.

```
public: logical ATTRIB_HH_AGGR_ISOSPLINE::
    make_c1 () const;
```

Gets the option to denote if C1 calculations are on or off.

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::make_strips_C1 ();
```

Makes the C1 surface strips.

```
public: virtual logical
    ATTRIB_HH_AGGR_ISOSPLINE::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::preprocess ();
```

Used for isospline healing. Locates inconsistencies in the topology, and should be used before other healing subprocesses.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::print (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the isospline solver subphase to the specified file.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::
    print_analyze (
    FILE* fp                // file pointer
    );
```

Prints the results of the analyze stage of the isospline solver subphase.

```
public: void
    ATTRIB_HH_AGGR_ISOSPLINE::print_calculate (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the calculate stage of the isospline solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_ISOSPLINE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if(restore_version_number >= TOL_MODELING_VERSION)
    read_logical                This is the save_sw data item.
```

```
protected: void
    ATTRIB_HH_AGGR_ISOSPLINE::set_C1_seq (
    ENTITY_LIST&            // entities
    );
```

Sets the face sequence for the C1 calculation.

```
public: void
    ATTRIB_HH_AGGR_ISOSPLINE::set_do_isospline (
    logical l                // use this subphase flag
    );
```

Sets a flag indicating whether or not the isospline solver subphase needs to be used. If TRUE, the isospline solver subphase is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::set_make_c1 (
    logical l                // make_c1
    );
```

Sets the make_c1 option.

```
public: void
    ATTRIB_HH_AGGR_ISOSPLINE::set_twist_correction (
    logical l                // twist correction
    );
```

Sets the twist_correction option on or off.

```
public: void ATTRIB_HH_AGGR_ISOSPLINE::sprint (
    char*                    // character string
    );
```

Prints statistics of the results of the isospline solver subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_ISOSPLINE::sprint_analyze (
    char*                    // character string
    );
```

Prints statistics of the results of the analyze stage of the isospline solver subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_ISOSPLINE::sprint_calculate (
    char*                    // character string
    );
```

Prints statistics of the results of the calculate stage of the isospline solver subphase to the specified string.

```
public: virtual const char*
    ATTRIB_HH_AGGR_ISOSPLINE::type_name () const;
```

Returns the string “aggregate_isospline_attribute”.

Related Fncs:

find_aggr_isospline, is_ATTRIB_HH_AGGR_ISOSPLINE

ATTRIB_HH_AGGR_SHARP_EDGE

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the sharp edge solver subphase of geometry building.

Derivation: ATTRIB_HH_AGGR_SHARP_EDGE : ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_sharped_attribute”

Filename: heal/healhusk/attrib/hsharped.hxx

Description: The ATTRIB_HH_AGGR_SHARP_EDGE class is attached to the body to be healed. Is is used by the sharp edge solver subphase of the geometry building healing phase. The sharp edge solver attempts to heal all edges and vertices that are shared by surfaces that intersect sharply. This includes nontangential surface junctions.

Limitations: None

References: None

Data:

`protected bhl_anal_geometry_results anal_results;`
Structure containing the results of the analyze stage of the sharp edge solver subphase (analysis of the input body; e.g., before healing).

`protected bhl_geombld_options opts;`
Structure containing the solver options.

`protected bhl_transversal_solver_results results;`
Structure containing the results of the sharp edge solver subphase.

```
protected hh_sharped_options sharped_opt;
```
Structure containing the options used by the sharp edge solver subphase. These options can be set and queried using methods of this class.

```
protected logical save_sw;
```
For future use.

Constructor:

```
public: ATTRIB_HH_AGGR_SHARP_EDGE::
    ATTRIB_HH_AGGR_SHARP_EDGE (
    BODY* b                  // owning body
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_SHARP_EDGE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_SHARP_EDGE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_SHARP_EDGE::
    ~ATTRIB_HH_AGGR_SHARP_EDGE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_SHARP_EDGE(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_SHARP_EDGE::analyze ();
```

Performs the analyze stage of the sharp edge solver subphase of geometry building. Analyzes the body and compute the best options and tolerances for the sharp edge solver subphase.

```
public: const bhl_anal_geometry_results
    ATTRIB_HH_AGGR_SHARP_EDGE::anal_geom_results
    () const;
```

Returns the results of geometry analysis (from analyze stage) of the input body (before healing).

```
public: virtual void
    ATTRIB_HH_AGGR_SHARP_EDGE::calculate ();
```

Performs the calculate stage of the sharp edge solver subphase of geometry building and stores all the recommended changes in individual attributes.

```
public: const bhl_transversal_solver_results
    ATTRIB_HH_AGGR_SHARP_EDGE::calculation_results
    () const;
```

Returns the results of the sharp edge solver subphase.

```
public: virtual void
    ATTRIB_HH_AGGR_SHARP_EDGE::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: logical ATTRIB_HH_AGGR_SHARP_EDGE::
    do_sharp_edge () const;
```

Returns FALSE if the sharp edge solver subphase has been turned off, otherwise, returns TRUE.

```
public: bhl_geombld_options*
    ATTRIB_HH_AGGR_SHARP_EDGE::
    get_current_results ();
```

Returns the current option results.

```
public: virtual int
    ATTRIB_HH_AGGR_SHARP_EDGE::identity (
    int                         // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_SHARP_EDGE_LEVEL .

```
public: virtual logical
    ATTRIB_HH_AGGR_SHARP_EDGE::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical
    ATTRIB_HH_AGGR_SHARP_EDGE::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_HH_AGGR_SHARP_EDGE::print (
    FILE* fp                    // file pointer
    );
```

Prints statistics of the results of the sharp edge solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_SHARP_EDGE::print_calculate (
    FILE* fp                    // file pointer
    );
```

Prints statistics of the results of the calculate stage of the sharp edge solver subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_SHARP_EDGE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if(restore_version_number >= TOL_MODELING_VERSION)
    read_logical                    This is the save_sw data item.
```

```
public: void
    ATTRIB_HH_AGGR_SHARP_EDGE::set_do_sharp_edge (
    logical l                // use this subphase flag
    );
```

Sets a flag indicating whether or not the sharp edge solver subphase needs to be used. If TRUE, the sharp edge solver subphase is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_SHARP_EDGE::sprint (
    char*                    // character string
    );
```

Prints statistics of the results of the sharp edge solver subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_SHARP_EDGE::sprint_calculate (
    char*                    // character string
    );
```

Prints statistics of the results of the calculate stage of the sharp edge solver subphase to the specified string.

```
public: virtual const char*
    ATTRIB_HH_AGGR_SHARP_EDGE::type_name () const;
```

Returns the string “aggregate_sharped_attribute”.

Related Fncs:

---

find_aggr_sharp_edge, is_ATTRIB_HH_AGGR_SHARP_EDGE

ATTRIB_HH_AGGR_SIMPLIFY

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the geometry simplification phase.

Derivation: ATTRIB_HH_AGGR_SIMPLIFY : ATTRIB_HH_AGGR_SIMPLIFY_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_simgeom_attribute”

Filename: heal/healhusk/attrib/aggrsimg.hxx

Description: The ATTRIB_HH_AGGR_SIMPLIFY class is attached to the body to be healed. Is is used by the geometry simplification healing phase.

Limitations: None

References: None

Data:

`protected double m_tol;`
Tolerance used by the geometry simplification phase.

`protected logical m_do_simplify;`
Flag indicating whether or not the geometry simplification phase should be performed. If TRUE, the geometry simplification phase is used; otherwise, it is not used.

`protected logical m_do_simplify_log;`
Flag indicating whether or not logging should be performed. If TRUE, logging is used; otherwise, it is not used.

`protected logical m_planes_only;`
Flag to denote whether or not only planes should be simplified. If TRUE, only plane surface types are simplified.

`protected struct bhl_geom_misc m_geom_misc_stats;`
Structure containing information about the model such as the number of faces, etc.

`protected struct bhl_geom_types input_geom_stats;`
Structure containing information about the input body (before geometry simplification).

`protected struct bhl_geom_types output_geom_stats;`
Structure containing information about the output body (after geometry simplification).

```
protected logical save_sw;
```
For future use.

Constructor:

```
public: ATTRIB_HH_AGGR_SIMPLIFY::
    ATTRIB_HH_AGGR_SIMPLIFY (
    BODY* owner             // owning body
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_SIMPLIFY(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_SIMPLIFY::
    ~ATTRIB_HH_AGGR_SIMPLIFY ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_SIMPLIFY(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::analyze ();
```

Analyzes all the faces of the owning body for geometry simplification. Sets appropriate value for the geometry simplification tolerance based on this analysis.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR_SIMPLIFY::attach_attrib (
    ENTITY*                 // entity to attach to
    );
```

Attaches an individual entity–level attribute to the given entity. This method chains individual attributes.

```
public: void ATTRIB_HH_AGGR_SIMPLIFY::
    attach_attribs_to_splines ();
```

Attaches individual entity–level attributes to all spline faces.

```
public: bhl_geom_misc&
ATTRIB_HH_AGGR_SIMPLIFY::bhl_geom_misc_stats ();
```

Retrieves the bhl_geom_misc_stats structure.

```
public: bhl_geom_types& ATTRIB_HH_AGGR_SIMPLIFY::
    bhl_geom_types_func ();
```

Retrieves the bhl_geom_types_func structure.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::calculate ();
```

Traverses all faces and adds individual entity–level attributes to faces having spline geometry. Then, for every spline surface, calculates (if possible) the simplified geometry within the specified tolerance. Stores the simplified geometry in the individual entity–level attributes.

```
public: virtual void ATTRIB_HH_AGGR_SIMPLIFY::
    cleanup ();
```

Removes all individual entity–level attributes from the entities of the owning body.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::detach_attrib (
    ENTITY*                 // owning entity
    );
```

Removes the entity–level attribute from the entity.

```
public: void
    ATTRIB_HH_AGGR_SIMPLIFY::detach_empty_attribs ();
```

Detaches all empty individual entity–level attributes; i.e., those attributes that do not carry any simplified geometry.

```
public: logical
    ATTRIB_HH_AGGR_SIMPLIFY::do_simplify () const;
```

Returns FALSE if the geometry simplification phase has been turned off, otherwise, returns TRUE.

```
public: logical ATTRIB_HH_AGGR_SIMPLIFY::
    do_simplify_log () const;
```

Gets the option to denote if simplify log generation is on or off.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::entity_list (
    ENTITY_LIST&            // list of entities
    ) const;
```

Gets all the entities chained to the owning body. These are entities to which individual attributes are attached.

```
public: virtual void ATTRIB_HH_AGGR_SIMPLIFY::fix ();
```

Applies (fixes) all the changes that are stored in individual attributes for the geometry simplification phase to the body. The old geometry is then stored in the attributes.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR_SIMPLIFY::get_attrib (
    ENTITY*                 // owning entity
    ) const;
```

Gets the entity–level attribute for the corresponding entity.

```
public: virtual int
    ATTRIB_HH_AGGR_SIMPLIFY::identity (
    int                       // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_SIMPLIFY_LEVEL.

```
public: virtual logical
    ATTRIB_HH_AGGR_SIMPLIFY::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_converted_analytics () const;
```

Returns the number of of splines converted to analytics.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_converted_cones () const;
```

Returns the number of of splines converted to cones.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_converted_cylinders () const;
```

Returns the number of of splines converted to cylinders.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_converted_planes () const;
```

Returns the number of splines converted to planes.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_converted_spheres () const;
```

Returns the number of of splines converted to spheres.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_converted_tori () const;
```

Returns the number of of splines converted to tori.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_expected_analytics () const;
```

Returns the number of expected spline–to–analytic conversions.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_expected_cones () const;
```

Returns the number of expected spline–to–cone conversions.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_expected_cylinders () const;
```

Returns the number of expected spline–to–cylinder conversions.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_expected_planes () const;
```

Returns the number of expected spline–to–plane conversions.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_expected_spheres () const;
```

Returns the number of expected spline–to–sphere conversions.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_expected_tori () const;
```

Returns the number of expected spline–to–tori conversions.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_final_splines () const;
```

Returns the number of spline surfaces remaining in the final body.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_analytics () const;
```

Returns the number of input analytics.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_cones () const;
```

Returns the number of input cones.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_cylinders () const;
```

Returns the number of input cylinders.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_planes () const;
```

Returns the number of input planes.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_spheres () const;
```

Returns the number of input spheres.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_splines () const;
```

Returns the number of input splines.

```
public: int ATTRIB_HH_AGGR_SIMPLIFY::
    num_input_tori () const;
```

Returns the number of input tori.

```
public: virtual logical
    ATTRIB_HH_AGGR_SIMPLIFY::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: logical ATTRIB_HH_AGGR_SIMPLIFY::
    planes_only () const;
```

Returns the value of the m_planes_only flag, which indicates whether or not only planes should be simplified. If TRUE, only plane surface types are simplified.

```
public: virtual void ATTRIB_HH_AGGR_SIMPLIFY::print (
    FILE* fp                // file pointer
    );
```

Prints the geometry simplification analysis results of both the input and the output body to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::print_analyze (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the analyze stage of the geometry simplification phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::print_calculate (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the calculate stage of the geometry simplification phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::print_fix (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the fix stage of the geometry simplification phase to the specified file.

```
public: void ATTRIB_HH_AGGR_SIMPLIFY::
    restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if(restore_version_number >= TOL_MODELING_VERSION)
    read_logical                    This is the save_sw data item.
```

```
public: void
    ATTRIB_HH_AGGR_SIMPLIFY::set_do_simplify (
    logical                 // use this phase flag
    );
```

Sets a flag indicating whether or not the geometry simplification phase needs to be used. If TRUE, the geometry simplification phase is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_SIMPLIFY::
    set_do_simplify_log (
    logical                 // simplify log
    );
```

Sets the option to denote if simplify log generation is on or off.

```
public: void ATTRIB_HH_AGGR_SIMPLIFY::
    set_planes_only (
    logical                 // planes only flag value
    );
```

Sets the value of the m_planes_only flag, which indicates whether or not only planes should be simplified. If TRUE, only plane surface types are simplified.

```
public: void ATTRIB_HH_AGGR_SIMPLIFY::set_tol (
    double                  // tolerance to use
    );
```

Sets the tolerance used by the geometry simplification phase.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::sprint (
    char*                    // character string
    );
```

Prints statistics of the results of the geometry simplification phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::sprint_analyze (
    char*                    // character string
    );
```

Prints statistics of the results of the analyze stage of the geometry simplification phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::sprint_calculate (
    char*                    // character string
    );
```

Prints statistics of the results of the calculate stage of the geometry simplification phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY::sprint_fix (
    char*                    // character string
    );
```

Prints statistics of the results of the fix stage of the geometry simplification phase to the specified string.

```
public: double ATTRIB_HH_AGGR_SIMPLIFY::tol () const;
```

Gets the current value of the geometry simplification tolerance.

```
public: virtual const char*
    ATTRIB_HH_AGGR_SIMPLIFY::type_name () const;
```

Returns the string “aggregate_simgeom_attribute”.

Related Fncs:

find_aggr_simplify, is_ATTRIB_HH_AGGR_SIMPLIFY

ATTRIB_HH_AGGR_SIMPLIFY_BASE

Class: Healing, SAT Save and Restore

Purpose: Base HEAL aggregate attribute class for the geometry simplification phase.

Derivation: ATTRIB_HH_AGGR_SIMPLIFY_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_simgeom_base_attribute”

Filename: heal/healhusk/attrib/agrsimbs.hxx

Description: ATTRIB_HH_AGGR_SIMPLIFY_BASE is the base geometry simplification aggregate attribute class from which other HEAL aggregate attribute classes used in the geometry simplification phase are derived. Aggregate attributes are attached to the body being healed to store information about the geometry simplification phase. Aggregate attributes also manage the individual attributes attached to entities of the body during geometry simplification.

Limitations: None

References: None

Data:

None

Constructor:

```
public: ATTRIB_HH_AGGR_SIMPLIFY_BASE::
    ATTRIB_HH_AGGR_SIMPLIFY_BASE (
    BODY* owner             // owning body to heal
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_SIMPLIFY_BASE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_SIMPLIFY_BASE::
    ~ATTRIB_HH_AGGR_SIMPLIFY_BASE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_SIMPLIFY_BASE(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::identity (
    int                     // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_SIMGEOM_BASE_LEVEL .

```
public: virtual logical
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::pattern_compatible
() const;
```

Returns TRUE if this is pattern compatible.

```
public: void
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data — This class does not save any data

```
public: virtual const char*
    ATTRIB_HH_AGGR_SIMPLIFY_BASE::type_name () const;
```

Returns the string “aggregate_simgeom_base_attribute”.

Related Fncs:

is_ATTRIB_HH_AGGR_SIMPLIFY_BASE

ATTRIB_HH_AGGR_STITCH

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the stitching phase.

Derivation: ATTRIB_HH_AGGR_STITCH : ATTRIB_HH_AGGR_STITCH_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_stitch_attribute”

Filename: heal/healhusk/attrib/aggrstch.hxx

Description: The ATTRIB_HH_AGGR_STITCH class is attached to the body to be healed. Is is used by the stitching healing phase.

Limitations: None

References: None

Data:

```
protected bhl_anal_stitch_results
    anal_stitch_results;
```
Structure containing the results of the analysis stage of the stitching phase.

```
protected logical invalid_stitch;
```
Flag indicating the body could not be stitched.

```
protected bhl_stitch_options stitch_opt;
```
Structure containing stitching options and tolerances.

```
protected logical m_do_stitch_log;
```
Option for logging changes to entities in a stitching module.

```
protected bhl_stitch_results m_stitch_results;
```
Structure containing the results of the stitching phase.

```
protected double m_max_tol;
```
Maximum tolerance used by the stitching phase.

```
protected double m_min_tol;
```
Minimum tolerance used by the stitching phase.

```
protected logical m_do_stitch;
```
Flag indicating whether or not the stitching phase should be performed. If TRUE, the stitching phase is used; otherwise, it is not used.

```
protected logical m_stepped;
```
Flag indicating whether or not to use stepped stitching. If TRUE, stepped stitching is attempted before incremental stitching. If FALSE, only incremental stitching is used.

```
protected logical m_stitch_performed;
```
Flag indicating whether or not stitching was performed.

```
protected logical save_sw;
```
For future use.

Constructor:

```
public: ATTRIB_HH_AGGR_STITCH::
    ATTRIB_HH_AGGR_STITCH (
    BODY* owner             // owning body
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_STITCH(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_HH_AGGR_STITCH::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_STITCH::
    ~ATTRIB_HH_AGGR_STITCH ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_STITCH(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::analyze ();
```

Performs the analyze stage of stitching. Analyzes and classifies the types of entities in the body, determines the number of each type of entity, and determines an appropriate “stitch box” to use for stitching. This method then sets the options and tolerances needed by stitching based on this analysis of the body.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR_STITCH::attach_attrib (
    ENTITY*                 // entity to attach to
    );
```

Attaches an individual entity–level attribute to the given entity. This method chains individual attributes.

```
protected: void ATTRIB_HH_AGGR_STITCH::
    attach_attribs_to_edges ();
```

Attaches individual entity–level attributes to all the edges in the owning body.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::calculate ();
```

This method performs the pairing of edges and populates the individual entity–level attributes to the edges. This method splits lumps into bodies, performs the stitching calculations on these bodies and stores the results, then merges the lumps back into the owner body.

```
public: void
    ATTRIB_HH_AGGR_STITCH::calculate_at_tol (
    double tol              // tolerance to use
    );
```

Traverses the body and adds individual entity–level attributes to edge pairs that are candidates for stitching at the specified tolerance.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::cleanup ();
```

Removes all individual entity–level attributes from the entities of the owning body.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::detach_attrib (
    ENTITY*                 // owning entity
    );
```

Removes the entity–level stitch attribute from the entity.

```
public: void ATTRIB_HH_AGGR_STITCH::
    detach_redundant_attribs ();
```

If an edge is already shared and doesn't need to be stitched, this method detaches its entity–level stitch attribute.

```
public: logical
    ATTRIB_HH_AGGR_STITCH::do_stitch () const;
```

Logical to denote if stitching is required.

```
public: logical
    ATTRIB_HH_AGGR_STITCH::do_stitch_log () const;
```

Returns FALSE if logging has been turned off, otherwise, returns TRUE.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::entity_list (
    ENTITY_LIST&            // list of entities
    ) const;
```

Gets all the entities chained to the owning body. These are entities to which individual attributes are attached.

```
public: virtual void ATTRIB_HH_AGGR_STITCH::fix ();
```

Applies (fixes) all the changes that are stored in individual attributes for the stitching phase to the body. The old topology is then stored in the attributes. This method actually performs the stitching of the edges that are paired up via the individual attributes.

```
public: virtual ATTRIB_HH_ENT*
    ATTRIB_HH_AGGR_STITCH::get_attrib (
    ENTITY*                 // owning entity
    ) const;
```

Gets the entity–level attribute for the corresponding entity.

```
public: bhl_stitch_options const
    ATTRIB_HH_AGGR_STITCH::get_stitch_options ();
```

Gets the structure which contains stitching options and tolerances.

```
public: bhl_stitch_results const
    ATTRIB_HH_AGGR_STITCH::get_stitch_results ();
```

Gets the structure which contains stitching results.

```
public: virtual int ATTRIB_HH_AGGR_STITCH::identity (
    int                     // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_STITCH_LEVEL.

```
public: void ATTRIB_HH_AGGR_STITCH::
    invalid_unshared_edges (
    ENTITY_LIST&            // list of edges
    ) const;
```

Returns the invalid unshared edges that require the user to manually stitch bad regions.

```
public: virtual logical
    ATTRIB_HH_AGGR_STITCH::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: double
    ATTRIB_HH_AGGR_STITCH::max_tol () const;
```

Gets the current value of the maximum stitching tolerance.

```
public: double
    ATTRIB_HH_AGGR_STITCH::min_tol () const;
```

Gets the current value of the minimum stitching tolerance.

```
public: int
    ATTRIB_HH_AGGR_STITCH::num_free_faces () const;
```

Returns the number of free faces after stitching analysis.

```
public: int ATTRIB_HH_AGGR_STITCH::
    num_invalid_unshared_edges () const;
```

Returns the number of invalid unshared edges formed after the stitch calculate stage.

```
public: int
    ATTRIB_HH_AGGR_STITCH::num_sheet_lumps () const;
```

Returns the number of sheet lumps.

```
public: int
    ATTRIB_HH_AGGR_STITCH::num_solid_lumps () const;
```

Returns the number of solid lumps.

```
public: int ATTRIB_HH_AGGR_STITCH::
    num_unshared_loops () const;
```

Returns the number of unshared loops that can probably be closed by a cover sheet.

```
public: int ATTRIB_HH_AGGR_STITCH::
    num_valid_unshared_edges () const;
```

Returns the number of the number of valid unshared edges formed after stitch calculation.

```
public: logical ATTRIB_HH_AGGR_STITCH::pair (
    EDGE* edge1,            // first edge
    EDGE* edge2,            // second edge
    double tol              // tolerance to use
    );
```

Attempts to pair up two given edges. Returns TRUE if it is possible to pair the two edges.

```
public: virtual logical
    ATTRIB_HH_AGGR_STITCH::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: virtual void ATTRIB_HH_AGGR_STITCH::print (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the stitching phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::print_analyze (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the analyze stage of the stitching phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::print_calculate (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the calculate stage of the stitching phase to the specified file.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::print_fix (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the fix stage of the stitching phase to the specified file.

```
public: void ATTRIB_HH_AGGR_STITCH::reset_cache ();
```

Reset cache in all edges.

```
public: void
    ATTRIB_HH_AGGR_STITCH::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if(restore_version_number >= TOL_MODELING_VERSION)
    read_logical                This is the save_sw data item.
```

```
public: const bhl_stitch_results
    ATTRIB_HH_AGGR_STITCH::results () const;
```

Returns the results of stitching (m_stitch_results structure).

```
public: void ATTRIB_HH_AGGR_STITCH::set_do_stitch (
    logical                 // new flag value
    );
```

Sets a flag indicating whether or not the stitching phase needs to be used. If TRUE, the stitching phase is used; otherwise, it is not used.

```
public: void
    ATTRIB_HH_AGGR_STITCH::set_do_stitch_log (
    logical                 // new flag value
    );
```

Sets a flag indicating whether or not is to be used. If TRUE, logging is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_STITCH::set_max_tol (
    double                  // tolerance to use
    );
```

Sets the new value of the maximum stitching tolerance.

```
public: void ATTRIB_HH_AGGR_STITCH::set_min_tol (
    double                  // tolerance to use
    );
```

Sets the new value of the minimum stitching tolerance.

```
public: void ATTRIB_HH_AGGR_STITCH::set_stepped (
    logical                 // new flag value
    );
```

Sets the value of the m_stepped flag, which indicates whether or not stepped stitching should be used. If TRUE, stepped stitching is performed before incremental. Otherwise, only incremental stitching is performed.

```
public: bhl_stitch_options*
    ATTRIB_HH_AGGR_STITCH::set_stitch_options ();
```

Sets the stitching options.

```
public: virtual void ATTRIB_HH_AGGR_STITCH::sprint (
    char*                   // character string
    );
```

Prints statistics of the results of the stitching phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::sprint_analyze (
    char*                   // character string
    );
```

Prints statistics of the results of the analyze stage of the stitching phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::sprint_calculate (
    char*                   // character string
    );
```

Prints statistics of the results of the calculate stage of the stitching phase to the specified string.

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH::sprint_fix (
    char*                   // character string
    );
```

Prints statistics of the results of the fix stage of the stitching phase to the specified string.

```
public: logical
    ATTRIB_HH_AGGR_STITCH::stepped () const;
```

Gets the value of the m_stepped flag, which indicates whether or not stepped stitching should be used. If TRUE, stepped stitching is performed before incremental. Otherwise, only incremental stitching is performed.

```
public: logical
    ATTRIB_HH_AGGR_STITCH::stitch_performed () const;
```

Returns TRUE if any stitching was performed.

```
protected: logical
    ATTRIB_HH_AGGR_STITCH::stitch_two_edges (
    ATTRIB_HH_ENT_STITCH_EDGE* att  // edge attribute
    );
```

Stitches two edges—one that is the owner of the input individual entity–level edge attribute, and the other that is its partner edge (as indicated in the edge attribute). Returns TRUE if successful.

```
public: int ATTRIB_HH_AGGR_STITCH::type () const;
```

Returns a number that indicates the type of body based on analysis.

1 Body is a solid
2 Body is a sheet
3 Body is very bad and needs user to intervene (e.g., manually stitch)

```
public: virtual const char* ATTRIB_HH_AGGR_STITCH::
    type_name () const;
```

Returns the string “aggregate_stitch_attribute”.

```
public: void ATTRIB_HH_AGGR_STITCH::
    valid_unshared_edges (
    ENTITY_LIST&            // list of edges
    ) const;
```

Returns a list of the unshared edges that are valid and can be closed by a cover sheet.

Related Fncs:

find_aggr_stitch, is_ATTRIB_HH_AGGR_STITCH

ATTRIB_HH_AGGR_STITCH_BASE

Class: Healing, SAT Save and Restore

Purpose: Base HEAL aggregate attribute class for the stitching phase.

Derivation: ATTRIB_HH_AGGR_STITCH_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_stitch_base_attribute”

Filename: heal/healhusk/attrib/agrstcbs.hxx

Description: ATTRIB_HH_AGGR_STITCH_BASE is the base stitch aggregate attribute class from which other HEAL aggregate attribute classes used in the stitching phase are derived. Aggregate attributes are attached to the body being healed to store information about the stitching phase. Aggregate attributes also manage the individual attributes attached to entities of the body during stitching.

Limitations: None

References: None

Data:

None

Constructor:

```
public: ATTRIB_HH_AGGR_STITCH_BASE::
    ATTRIB_HH_AGGR_STITCH_BASE (
    BODY* owner              // owning body to heal
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_STITCH_BASE(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH_BASE::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_STITCH_BASE::
    ~ATTRIB_HH_AGGR_STITCH_BASE ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_STITCH_BASE(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_STITCH_BASE::debug_ent (
    FILE*                    // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: virtual int
    ATTRIB_HH_AGGR_STITCH_BASE::identity (
    int                      // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_STITCH_BASE_LEVEL.

```
public: virtual logical
    ATTRIB_HH_AGGR_STITCH_BASE::is_deepcopyable ()
const;
```

Returns TRUE if this can be deep copied.

```
public: virtual logical
    ATTRIB_HH_AGGR_STITCH_BASE::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: void
    ATTRIB_HH_AGGR_STITCH_BASE::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

No data — This class does not save any data

```
public: virtual const char*
     ATTRIB_HH_AGGR_STITCH_BASE::type_name () const;
```

Returns the string “aggregate_stitch_base_attribute”.

Related Fncs:

is_ATTRIB_HH_AGGR_STITCH_BASE

ATTRIB_HH_AGGR_WRAPUP

Class: Healing, SAT Save and Restore

Purpose: Aggregate healing attribute class for the wrap–up subphase of geometry building.

Derivation: ATTRIB_HH_AGGR_WRAPUP : ATTRIB_HH_AGGR_GEOMBUILD_BASE : ATTRIB_HH_AGGR : ATTRIB_HH : ATTRIB : ENTITY : ACIS_OBJECT : –

SAT Identifier: “aggregate_secndry_attribute”

Filename: heal/healhusk/attrib/hsecndry.hxx

Description: The ATTRIB_HH_AGGR_WRAPUP class is attached to the body to be healed. Is is used by the wrap–up subphase of the geometry building healing phase. The wrap–up subphase recomputes the pcurve geometry of unhealed coedges (also referred to as “secondary geometries”). It trims curves, fixes pcurves, and orients normals.

Limitations: None

References: None

Data:

```
protected hh_secondary_solver_options options;
```
Structure containing the options used by the wrap–up subphase. The options can be set and queried using methods of this class.

```
protected bhl_wrapup_results results;
```
Structure containing results of the wrap–up subphase.

```
protected logical fix_enum;
```
Option for setting enums in edges to tangent.

```
protected logical save_sw;
```
For future use.

Constructor:

```
public: ATTRIB_HH_AGGR_WRAPUP::
    ATTRIB_HH_AGGR_WRAPUP (
    BODY* b                  // owning body
        = NULL
    );
```

C++ initialize constructor requests memory for this object and populates it with the data supplied as arguments. Applications should call this constructor only with the overloaded new operator inherited from the ENTITY class (for example, x=new ATTRIB_HH_AGGR_WRAPUP(...)), because this reserves the memory on the heap, a requirement to support roll back and history management.

Destructor:

```
public: virtual void ATTRIB_HH_AGGR_WRAPUP::lose ();
```

Posts a delete bulletin to the bulletin board indicating the instance is no longer used in the active model. The lose methods for attached attributes are also called.

```
protected: virtual ATTRIB_HH_AGGR_WRAPUP::
    ~ATTRIB_HH_AGGR_WRAPUP ();
```

This C++ destructor should never be called directly. Instead, applications should use the overloaded lose method inherited from the ENTITY class, because this supports history management. (For example, x=new ATTRIB_HH_AGGR_WRAPUP(...) then later x–>lose.)

Methods:

```
public: virtual void
    ATTRIB_HH_AGGR_WRAPUP::analyze ();
```

Performs the analyze stage of the wrap–up subphase of geometry building. Analyzes the body and compute the best options for the wrap–up subphase.

```
public: virtual void
    ATTRIB_HH_AGGR_WRAPUP::calculate ();
```

Performs the calculate stage of the wrap–up subphase of geometry building and stores all the recommended changes in individual attributes.

```
public: const bhl_wrapup_results
    ATTRIB_HH_AGGR_WRAPUP::calculation_results
    () const;
```

Returns the results of the wrap–up subphase.

```
public: virtual void
    ATTRIB_HH_AGGR_WRAPUP::debug_ent (
    FILE*                   // file pointer
    ) const;
```

Prints the type and address of this object, roll back pointer, attributes, and any unknown subtype information to the specified file. Refer to the ENTITY class for more details.

```
public: logical
    ATTRIB_HH_AGGR_WRAPUP::do_enum () const;
```

Returns FALSE if the fix_enum option has been turned off, otherwise, returns TRUE.

```
public: logical ATTRIB_HH_AGGR_WRAPUP::do_wrapup ();
```

Returns FALSE if the wrap–up subphase has been turned off, otherwise, returns TRUE.

```
public: virtual int ATTRIB_HH_AGGR_WRAPUP::identity (
    int                     // derivation level
        = 0
    ) const;
```

If level is unspecified or 0, returns the type identifier ATTRIB_HH_TYPE. If level is specified, returns <class>_TYPE for that level of derivation from ENTITY. The level of this class is defined as ATTRIB_HH_AGGR_WRAPUP_LEVEL.

```
public: virtual logical
    ATTRIB_HH_AGGR_WRAPUP::is_deepcopyable () const;
```

Returns TRUE if this can be deep copied.

```
public: int
    ATTRIB_HH_AGGR_WRAPUP::num_bad_coedges ();
```

Returns the number of bad coedges.

```
public: virtual logical
    ATTRIB_HH_AGGR_WRAPUP::pattern_compatible ()
const;
```

Returns TRUE if this is pattern compatible.

```
public: void ATTRIB_HH_AGGR_WRAPUP::print (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the wrap–up subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_WRAPUP::print_analyze (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the analyze stage of the wrap–up subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_WRAPUP::print_calculate (
    FILE* fp                // file pointer
    );
```

Prints statistics of the results of the calculate stage of the wrap–up subphase to the specified file.

```
public: void
    ATTRIB_HH_AGGR_WRAPUP::restore_common ();
```

The RESTORE_DEF macro expands to the restore_common method, which is used in reading information from a SAT file. This method is never called directly. It is called by a higher hierarchical function if an item in the SAT file is determined to be of this class type. An instance of this class will already have been created through the allocation constructor. This method then populates the class instance with the appropriate data from the SAT file.

```
if(restore_version_number >= TOL_MODELING_VERSION)
    read_logical                    This is the save_sw data item.
```

```
public: bhl_wrapup_results&
ATTRIB_HH_AGGR_WRAPUP::results_for_change ();
```

Gets the structure which contains the results for wrapup, such as the number of pcurves calculated etc.

```
public: void ATTRIB_HH_AGGR_WRAPUP::set_do_wrapup (
    logical use             // new flag value
    );
```

Sets a flag indicating whether or not the wrap–up subphase needs to be used. If TRUE, the wrap–up subphase is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_WRAPUP::set_enum (
    logical use             // new flag value
    );
```

Sets a flag indicating whether or not the fix_enum option needs to be used. If TRUE, the fix_enum option is used; otherwise, it is not used.

```
public: void ATTRIB_HH_AGGR_WRAPUP::sprint (
    char*                   // character string
    );
```

Prints statistics of the results of the wrap–up subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_WRAPUP::sprint_analyze (
    char*                   // character string
    );
```

Prints statistics of the results of the analyze stage of the wrap–up subphase to the specified string.

```
public: void
    ATTRIB_HH_AGGR_WRAPUP::sprint_calculate (
    char*                   // character string
    );
```

Prints statistics of the results of the calculate stage of the wrap–up subphase to the specified string.

```
public: virtual const char* ATTRIB_HH_AGGR_WRAPUP::
    type_name () const;
```

Returns the string “aggregate_secndry_attribute”.

Related Fncs:

find_aggr_wrapup, is_ATTRIB_HH_AGGR_WRAPUP